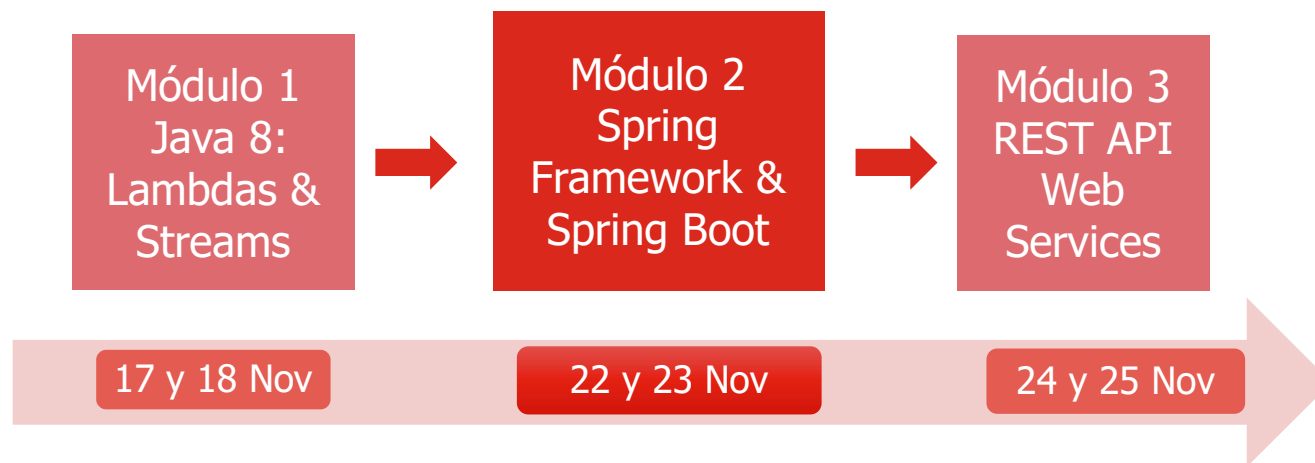


Programa Juniors Backend GFT



Formador: Ezequiel Llarena Borges

Bloques principales de contenidos

Módulo 2
Spring
Framework &
Spring Boot

Spring Core

Spring MVC

Spring Data
JPA

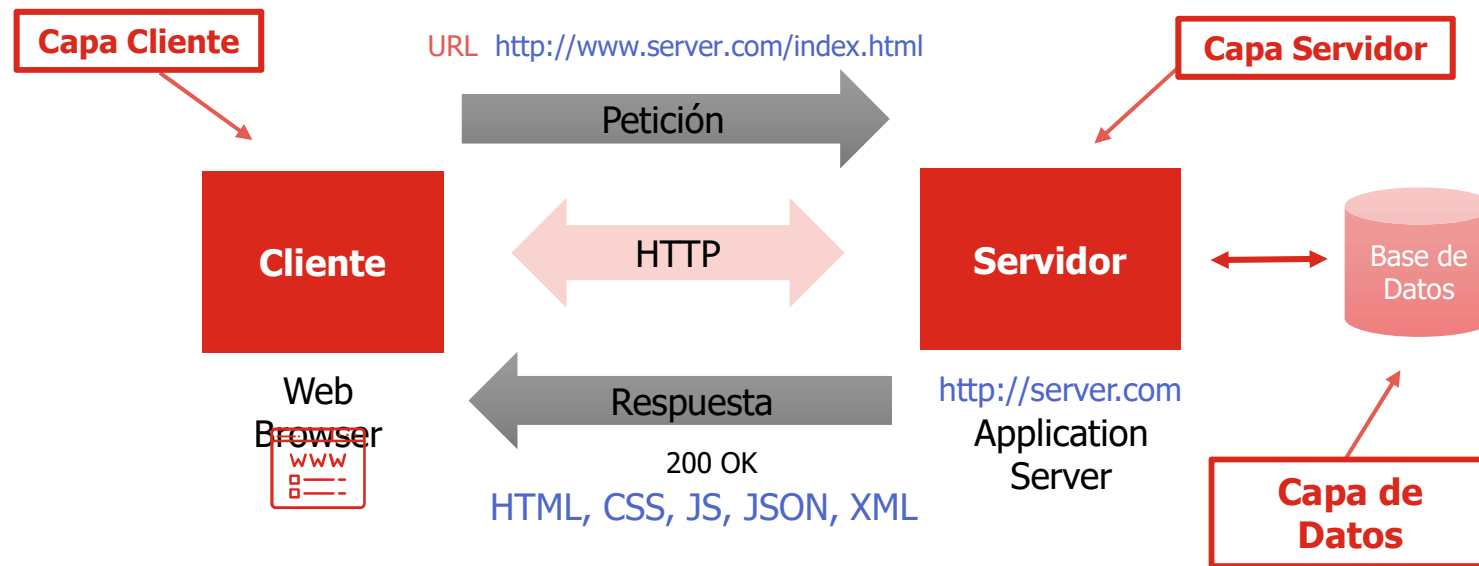
Spring Boot

Introducción

Arquitectura en capas

Arquitectura web

Modelo de arquitectura en capas



Tecnologías de la capa Servidor

- Permiten crear e implementar una aplicación web en el lado del servidor:



- **Java:** uno de los lenguajes de programación más utilizados para desarrollar el backend de una aplicación web.
- **Node JS:** entorno de ejecución multiplataforma de código abierto del lado del servidor basado en el lenguaje de programación JavaScript.
- **PHP:** lenguaje de programación con licencia libre, multiplataforma y se integra con Apache y MySQL. Actualmente ha perdido popularidad en su utilización.
- **Python:** lenguaje de programación multiparadigma para desarrollar software multiplataforma utilizado en Big Data, videojuegos, web scraping, Inteligencia Artificial y acceso a APIs.
- Django, Groovy, ASP .NET, Java EE,...

Tecnologías de la capa de Datos

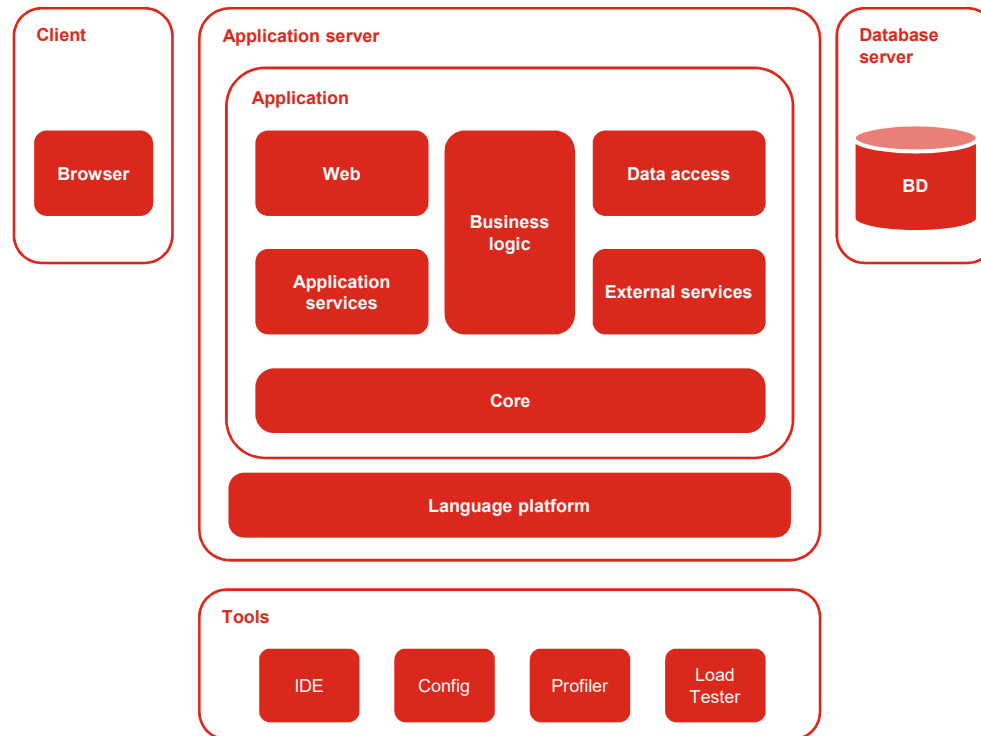
- Sistemas de persistencia de datos e información manejada por la aplicación web:



- **Bases de datos relacionales:** MySQL, Oracle, MS SQL Server, PostgreSQL...
- **Bases de datos no relacionales:** Mongo DB, Cassandra, riak, redis...

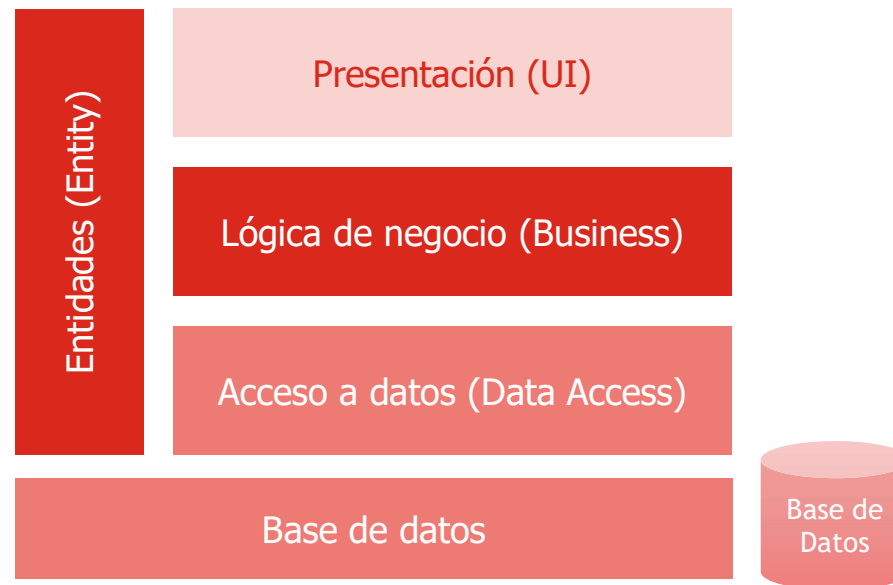
Arquitecturas web

Capas en arquitectura de ejecución



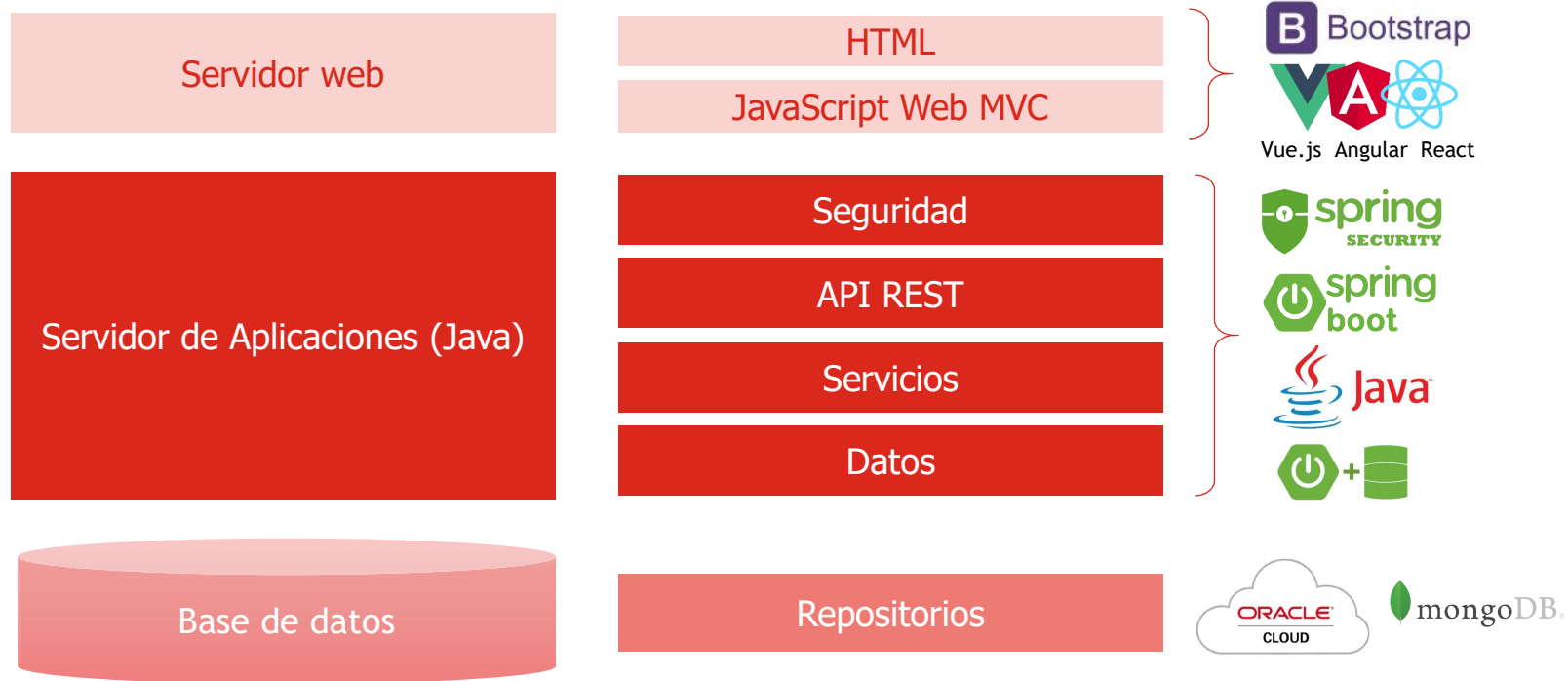
Arquitectura de n Capas

Desarrollo de aplicaciones basado en capas



Arquitectura de n Capas

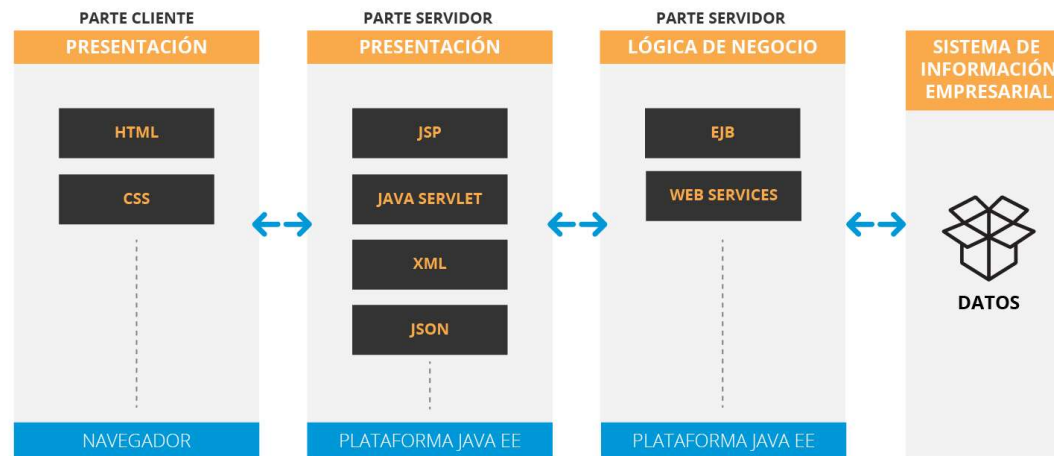
Desarrollo de aplicaciones basado en capas



Arquitecturas web

Servidores de aplicaciones

- Un **servidor de aplicaciones** es un **contenedor** que abarca la **lógica de negocio** de un sistema, y que **provee respuestas a las peticiones** de distintos dispositivos que tienen acceso a ella.
- Incluyen middleware que les permite intercomunicarse con otros servicios, para efectos de confiabilidad, seguridad y no-repudio.
- Brindan a los desarrolladores una Interfaz para Programación de Aplicaciones (API).
- Soporte a variedad de estándares: HTML, XML, IIOP, JDBC, SSL, etc.



Servidor de aplicaciones

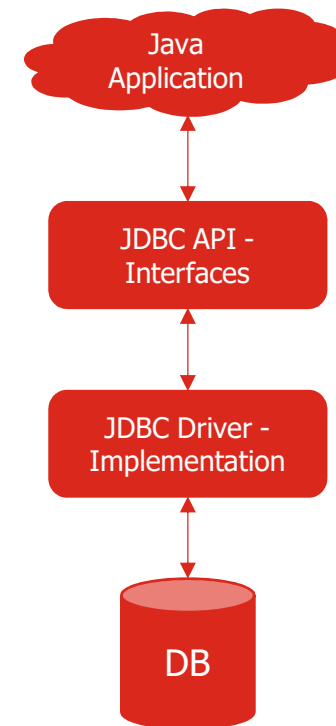
Java Enterprise

- Producto que implementa las APIs especificadas en Java EE.
- Permite configurar servicios que incluyen algunos de los estándares y desplegar aplicaciones web que tienen componentes que implementan estándares, como JSP, Servlet o JSF.
- Permite desplegar aplicaciones que publican servicios web a través de puertos, que pueden ser accedidos por URL desde un browser o desde aplicaciones cliente.
- Corporativos de pago:
 - WebLogic de Oracle
 - IBM WebSphere
 - JBoss Enterprise Platform, de Red Hat
- Corporativos libres:
 - GlassFish, de Oracle (heredado de Sun).
 - JBoss Application Server.
 - WebSphere Community Edition
- Libres:
 - TomEE, de Apache. Versión Java EE de Tomcat.
 - JOnAS, de Object Web.
 - Geronimo, de Apache.

Transacciones y Persistencia

Persistencia de datos con JDBC

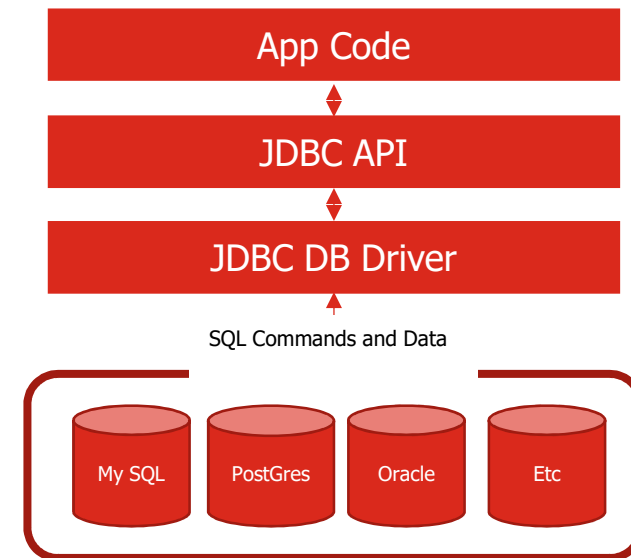
- La interfaz **JDBC (Java Data Base Connectivity)** es una capa de abstracción que ayuda a las aplicaciones a usar comandos SQL sin preocuparse por la implementación real de la base de datos.
- JDBC es un conjunto de **clases e interfaces** Java para ejecutar **sentencias SQL** sobre una base de datos.
- Ofrece un **estándar de conexión** a cualquier base de datos disponible en el mercado.
- Permite obtener los datos de forma fácil y cómoda en ambientes cliente-servidor a través de internet/intranet.



Transacciones y Persistencia

Persistencia de datos con JDBC

- JDBC consta de dos capas:
 - La **API JDBC** admite la comunicación entre la aplicación Java y el administrador JDBC.
 - El **controlador JDBC** admite la comunicación entre el administrador JDBC y el controlador de la base de datos. Es un clase que implementa la API JDBC para una base de datos específica.



Transacciones y Persistencia

Persistencia de datos con JDBC

- Pasos para conectarse a una base de datos con JDBC:
 - Instalar la base de datos con la que queremos trabajar.
 - Incluir la biblioteca JDBC al proyecto.
 - Añadir el controlador JDBC a nuestro classpath.
 - Utilizar la biblioteca JDBC para obtener una conexión a la base de datos.
 - Utilizar la conexión para ejecutar comandos SQL.
 - Cerrar la conexión a la base de datos cuando hayamos terminado.

Transacciones y Persistencia

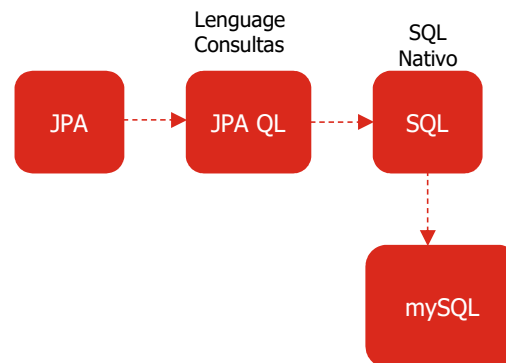
Persistencia de datos con JDBC

- Clases del paquete `java.sql` que facilitan la conexión estándar de la base de datos Java:
 - `Connection` representa la conexión a la base de datos.
 - `DriverManager` obtiene la conexión a la base de datos (otra opción es `DataSource` para la agrupación de conexiones).
 - `SQLException` maneja los errores de SQL entre la aplicación Java y la base de datos.
 - `ResultSet` y `Statement` modelan los conjuntos de resultados de datos y las declaraciones SQL.

Transacciones y Persistencia

Java Persistence API (JPA)

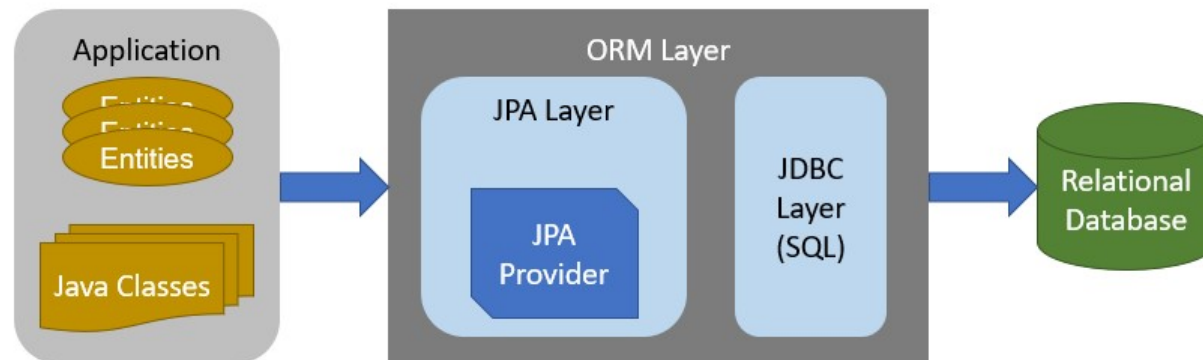
- **API de persistencia** desarrollada por la plataforma Java EE.
- Estándar Java para implementar un framework Object Relational Mapping (ORM), que permite interactuar con la base de datos por medio de objetos. JPA es el encargado de convertir los objetos Java en instrucciones para el Manejador de Base de Datos (MDB).
- Es un **framework** del lenguaje Java que permite manejar **datos relacionales** en aplicaciones usando plataformas Java en sus ediciones JSE y JEE. Definida en **javax.persistence**.



Transacciones y Persistencia

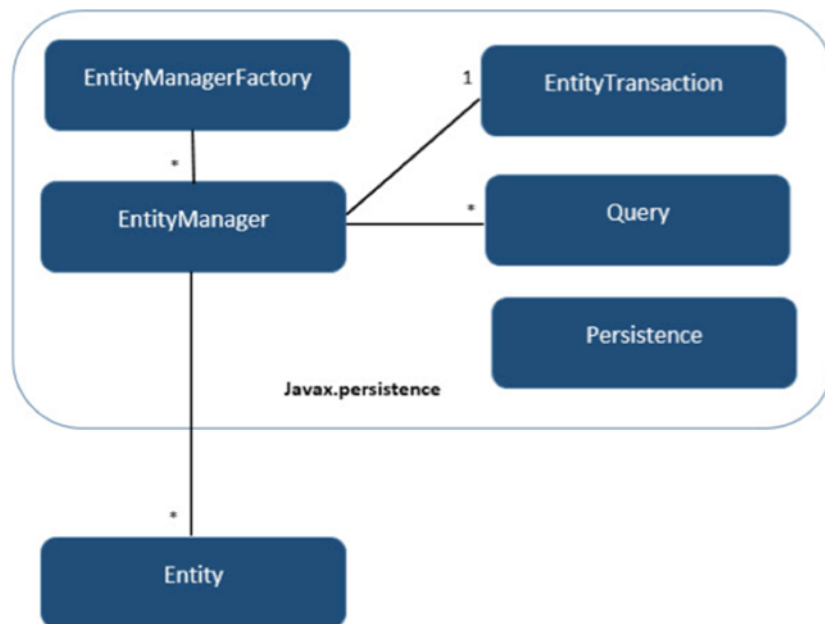
Persistencia de objetos con JPA

- JPA especifica cómo acceder, conservar y administrar los **datos entre los objetos de Java y la base de datos relacional**.
- JPA es la propuesta estándar que ofrece Java para implementar un framework **Object Relational Mapping (ORM)**, que permite interactuar con la base de datos por medio de objetos.
- La capa de mapeo relacional de objetos (ORM) es responsable de **convertir objetos** para interactuar **con tablas y columnas** en una base de datos relacional. En Java, las clases y objetos de Java se convierten mediante la capa ORM para que puedan persistir en una base de datos relacional.



Transacciones y Persistencia

Clases principales e interfaces de la arquitectura JPA



- **EntityManager**: interfaz que gestiona la persistencia de objetos. Funciona como instancia de consulta.
- **EntityManagerFactory**: clase factoría de **EntityManager**. Crea y gestiona múltiples instancias **EntityManager**.
- **Entity**: las entidades son los objetos de persistencia, tantas como registros en la base de datos.
- **EntityTransaction**: tiene una relación de uno a uno con **EntityManager**. Para cada método **EntityManager**, se mantienen las operaciones de **EntityTransaction** clase.
- **Persistence**: contiene métodos estáticos para obtener **EntityManagerFactory**.
- **Query**: interfaz implementada por cada proveedor JPA relacional para obtener objetos que cumplan los criterios.

Spring


Spring Framework & Spring Boot

Spring Framework Características

- Conjunto de reglas y herramientas destinadas a reducir los tiempos de desarrollo de aplicaciones Java
- Marco de trabajo más utilizado y más popular del ecosistema Java
- Open Source
- Arquitectura MVC
- Uso de Anotaciones
- Permite Programación Orientada a Aspectos (AspectJ)
- Principalmente enfocado a aplicaciones empresariales en Java EE (además soporte para Groovy y Kotlin)
- Especialmente útil para desarrollar servicios web sobre REST API
- Módulos de Testing para pruebas unitarias y de integración
- Gestión programática de transacciones

Principios en los que se basa Spring

- Inyección de Dependencias (DI)
- Programación orientada a aspectos (AOP)
- Plantillas
- Alta cohesión
- Bajo Acoplamiento
- Bean



Los **beans** son la manera que tiene de denominar Spring a los **objetos Java de los que se encarga**, es decir aquellos que se encuentren en el **contenedor** de Spring

Beneficios de Spring

- Simplifica aplicaciones JEE (POJO, Java Bean)
- Flexibilidad (Integración con otras herramientas)
- Framework ligero, no intrusivo
- Reduce los tiempos y el coste de los desarrollos
- Evita el tener que picar gran parte del código o de realizar tediosas configuraciones
- Reduce código repetitivo (JDBC, ...)
- Reduce la complejidad del desarrollo de una aplicación con altas complejidades
- Facilita la automatización de muchos procesos (acceso a BBDD, inyección de dependencias...)

Spring

Módulos y Proyectos

Estructura de Proyectos Spring

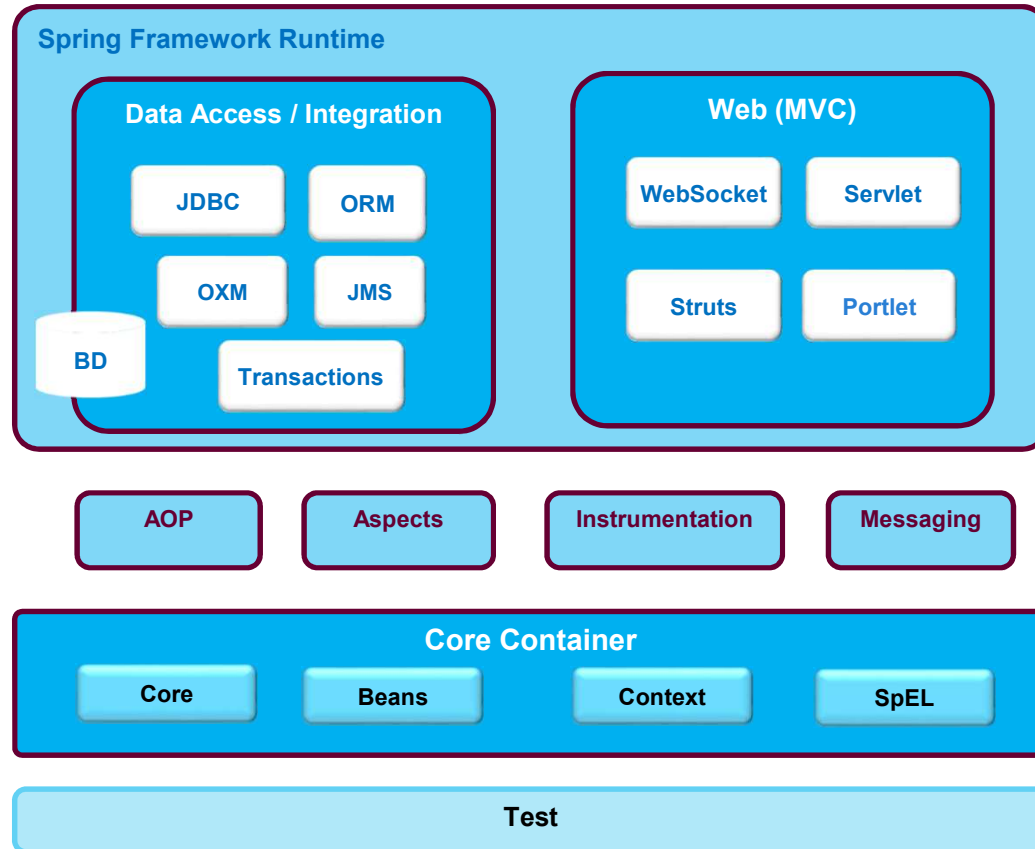
MÓDULO

- Funcionalidad relacionada con el contenedor al que engloba

PROYECTO

- Agrupa conjunto de módulos por funcionalidades o tecnologías

Módulos



Spring **Core Container**

- Contenedor de Spring
- Fábrica de beans de Spring
- ApplicationContext
- SpEL (Spring Expression Language)
- Objeto basado en patrón de diseño Factory
- Es un IoC container
- Gestiona ciclo de vida de los objetos (beans)
 - ✓ Crearlos
 - ✓ Enlazarlos
 - ✓ Configurarlos
 - ✓ Destruirlos

AOP e Instrumentación

- Programación Orientada a Aspectos
- Instrumentalizar clases Java (ClassLoader)
- Módulo spring-aop
- Módulo spring-aspects (AspectJ)

Data Access/Integration

- Controlar acceso a datos
- Capa de abstracción para acceder a BD
- Integración con bases datos relacionales y ORM
- Módulo spring-jdbc
- Módulo spring-orm
- Módulo spring-tx para gestión programática de transacciones para todas las clases y POJO
- Módulo spring-oxm (capa de abstracción para el mapeo de objetos o de XML)
- Módulo spring-jms para la producción y el procesamiento de mensajes.

Web

- Módulo específico para aplicaciones web
- Servicios web REST
- spring-web
- spring-webmvc (web servlet)
- spring-websocket

Test

- Componentes para hacer pruebas
- Pruebas Unitarias
- Pruebas de Integración
- Junit o TestNG
- Módulo spring-test

Proyectos Spring

- Spring Boot
- Spring Framework
- Spring Data
- Spring Cloud
- Spring Cloud Data Flow
- Spring Security
- Spring Session
- Spring Integration
- Spring HATEOAS
- Spring REST Docs
- Spring Batch
- Spring Web Services

spring.io/projects

- Spring AMQP
- Spring for Android
- Spring CredHub
- Spring Flo
- Spring for Apache Kafka
- Spring LDAP
- Spring Mobile
- Spring Roo
- Spring Shell
- Spring Statemachine
- Spring Vault
- Spring Web Flow

Principales Proyectos Spring

Spring Framework	Core support for dependency injection, transaction management, web apps, data access, messaging, and more.
Spring Boot	Takes an opinionated view of building Spring applications and gets you up and running as quickly as possible.
Spring Data	Consistent approach to data access – relational, non-relational, map-reduce, and beyond.
Spring Cloud	Set of tools for common patterns in distributed systems. Useful for building and deploying microservices.
Spring Security	Protects applications with comprehensive and extensible authentication and authorization support.
Spring Cloud Data Flow	Provides an orchestration service for composable data microservice applications on modern runtimes.

Principales Proyectos Spring

Spring Session	API and implementations for managing a user's session information.
Spring Integration	Supports the well-known Enterprise Integration Patterns through lightweight messaging and declarative adapters.
Spring HATEOAS	Simplifies creating REST representations that follow the HATEOAS principle.
Spring REST Docs	Lets you document RESTful services by combining hand-written documentation with auto-generated snippets produced with Spring MVC Test or REST Assured.
Spring Batch	Simplifies and optimizes the work of processing high-volume batch operations.
Spring Shell	Makes writing and testing RESTful applications easier with CLI-based resource discovery and interaction.

Spring

Gestión de Dependencias

Conceptos clave

- Acoplamiento
- Cohesión
- Contenedor de Inversión de Control (IoC)
- Inyección de dependencias

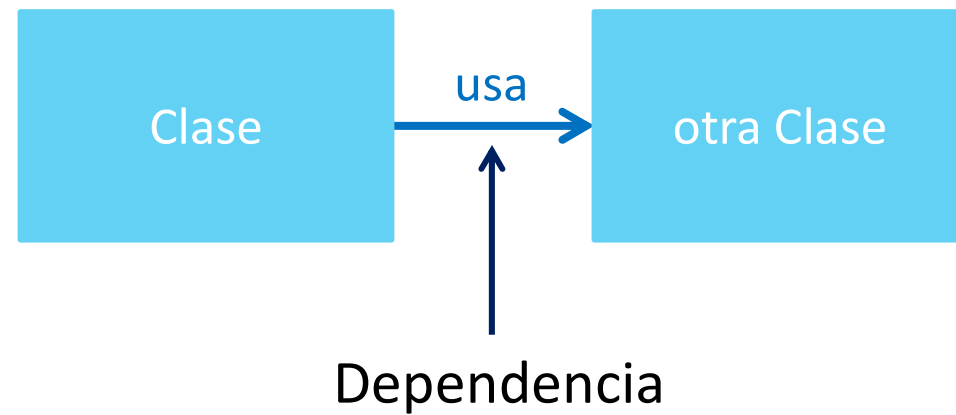
Principio de Inversión de Dependencia (DIP)

- Los módulos de alto nivel no deben depender de los módulos de bajo nivel
- Abstracciones sobre implementaciones
- Nos ayudará a crear código desacoplado
- El patrón IoC es una aplicación de este principio

Inversión de Control (IoC)


- Implementación del DIP (Dependency Inversion Principle)
- Principio de Hollywood (“No nos llames, nosotros te llamamos”)
- Estilo de programación donde un agente externo controla el flujo de la aplicación
- Término genérico que se implementa de distintas maneras:
 - ✓ Service Locator (otro patrón de diseño de software)
 - ✓ Events
 - ✓ Delegates
 - ✓ **Inyección de dependencias (DI)**

Inyección de Dependencias



Dependencia

```
public class Customer {  
    public Logger log;  
    public Customer() {  
        log = new Logger();  
    }  
}
```




Instancias de
Customer
fuertemente
acopladas a la
clase Logger

Desacoplar

```
public class Customer {  
    public Logger log;  
    public Customer(Logger obj) {  
        log = obj;  
    }  
}
```

Inyección de
Dependencias en
el Constructor



Inyección de Dependencias

- Patrón de Diseño de Software
- Elimina la dependencia entre dos clases
- Crea automáticamente instancias de una clase
- Inyección de dependencia con Reflection
- Subtipo de IoC
- Objetivo: código fácil de mantener
- Provee a los objetos lo que el objeto necesita:
 - ✓ Constructor
 - ✓ Propiedad (set)
 - ✓ Servicio / interfaz

Spring

Configuración de Spring

Configuración de Beans

- XML
- Anotaciones @
- JavaConfig

Ventajas
Inconvenientes

ID XML en el Constructor

```
<!-- default example (autowire="no") -->

<bean id="bean_id" class="com.example.SimpleSPImpl">
    <constructor-arg name="ds" ref="datasource" />
    <constructor-arg name="spName" value="spNameTest" />
</bean>

<!-- Definición de la dependencia de 'bean_id' -->
<bean id="datasource" class="com.mchange.ComboPooledDataSource">
    ...
</bean>
```

c-namespace

```
<bean id="bean_id" class="com.example.SimpleSPIImpl">  
    <constructor-arg name="ds" ref="datasource" />  
    <constructor-arg name="spName" value="spNameTest" />  
</bean>
```

```
<bean id="bean_id" class="com.example.SimpleSPIImpl"  
    c:ds-ref="datasource"  
    c:spName="spNameTest" />
```

```
<!-- Definición del datasource de ejemplo -->  
<bean id="datasource" class="com.mchange.ComboPooledDataSource">  
    ...  
</bean>
```

ID XML en el Setter

```
<!-- La propiedad debe llamarse igual que en el fuente Java -->

<bean id="driver" class="com.spring.domain.Driver">
  <property name="license" ref="license"/>
</bean>

<bean id="license" class="com.spring.domain.License" >
  <property name="number" value="123456ABCD"/>
</bean>
```

p-namespace

```
<!-- Service Beans -->
<bean id="bookService" class="com.bookstore.services.BookService"
    p:bookId="book1"
    p:bookDao-ref="bookDao" />















<bean id="purchasingService"
    class="com.bookstore.services.PurchasingService"
    p:bookServiceInterface-ref="bookService"
    p:accountServiceInterface-ref="accountService"></bean>

<bean id="accountService" class="com.bookstore.services.AccountService"
    p:accountDao-ref="accountDao" />
```

namespaces p y c en STS

Namespaces

Configure Namespaces
Select XSD namespaces to use in the configuration file

<input type="checkbox"/>		aop - http://www.springframework.org/schema/aop
<input checked="" type="checkbox"/>		beans - http://www.springframework.org/schema/beans
<input checked="" type="checkbox"/>		c - http://www.springframework.org/schema/c 
<input type="checkbox"/>		cache - http://www.springframework.org/schema/cache
<input type="checkbox"/>		context - http://www.springframework.org/schema/context
<input type="checkbox"/>		jdbc - http://www.springframework.org/schema/jdbc
<input type="checkbox"/>		jee - http://www.springframework.org/schema/jee
<input type="checkbox"/>		jms - http://www.springframework.org/schema/jms
<input type="checkbox"/>		lang - http://www.springframework.org/schema/lang
<input type="checkbox"/>		p - http://www.springframework.org/schema/p
<input type="checkbox"/>		task - http://www.springframework.org/schema/task
<input type="checkbox"/>		tx - http://www.springframework.org/schema/tx
<input type="checkbox"/>		util - http://www.springframework.org/schema/util

Spring Bean Autowiring (XML)

```
<!--Using autowire attribute in <bean> tag. shapeBox wire by name -->  
<bean id="shapeBox2" class="com.beans.ShapeBox" autowire="byName" />  
  
<!--shapeBox wire by type -->  
<bean id="shapeBox3" class="com.beans.ShapeBox" autowire="byType" />  
  
<!--shapeBox wire by constructor -->  
<bean id="shapeBox4" class="com.beans.ShapeBox" autowire="constructor"/>
```

spring-config.xml

Formas de Configuración de Beans

- XML
- **@notaciones**
- JavaConfig

Escaneo de Anotaciones

Spring navegará por las clases buscando las anotaciones:

```
<context:component-scan base-package="com.comp.beans"></context:component-scan>
```

```
<!-- necesario para que busque beans en la ruta indicada por base-package -->
```

Tipos de Beans

- Presentación
- Lógica de Negocio / Servicio
- Acceso a Datos

Estereotipos configurables

- @Component
- @Service
- @Repository
- @Controller

Autowiring

Permite **resolver la inyección de dependencias** de los siguiente modos:

- En el constructor de la clase
- En un atributo
- En un método setter
- En un método JavaConfig (**autowire="byName"**)

@Autowired en el constructor

La inyección se realiza en el momento en que el objeto es creado.

```
@Component
public class MyController {

    private final MyBean myBean;

    @Autowired
    public MyController(MyBean myBean) {
        this.myBean = myBean;
    }
}
```

@Autowired en el constructor (con @Value)

La inyección se realiza en el momento en que el objeto es creado.

```
@Controller
public class MyController {

    //@Value("1")
    private int id;
    private MyBean myBean;

    @Autowired
    public MyController(@Value("1")int id, MyBean myBean) {
        this.myBean = myBean;
    }
}
```

@Autowired en el setter

Se creará el método y una vez creado, Spring inyectará el *bean* mediante dicho método.

```
@Controller
public class MyController {

    private MyBean myBean;

    @Autowired
    public void setMyBean(MyBean myBean) {
        this.myBean = myBean;
    }
}
```


@Value en el setter

Uso de @Value para manejar parámetros en el constructor:

```
@Component
public class Direccion {

    private String calle;

    @Autowired
    public void setCalle(@Value("Calle Amparo")String calle) {
        this.calle = calle;
    }
}
```

@Autowired en un atributo

Spring crea la instancia del objeto y una vez creada le inyecta la dependencia.

```
@Controller
public class MyController {

    @Autowired
    private MyBean myBean;
}
```

Dependency checking

```
import org.springframework.beans.factory.annotation.Autowired;
```

```
public class Customer {
```

```
    // Indicamos que no hay que satisfacer la dependencia
```

```
    @Autowired(required=false)
```

```
    private Person person;
```

```
    private int type;
```

```
    private String action;
```

```
    //getter and setter methods }}
```

@Qualifier

```
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;
```

```
public class Customer {  
    @Autowired  
    @Qualifier("personBean1")  
    private Person person;  
    private int type;  
    private String action;  
  
    //getter and setter methods }  
}
```

Formas de Configuración de Beans

- XML
- @notaciones
- **JavaConfig**

JavaConfig

```
@Configuration  
public class AppConfig {  
  
    @Bean  
    public MyService myService() {  
        return new MyServiceImpl();  
    }  
}
```

Equivalente a definir:

<bean...> en XML

Otras anotaciones estándar JEE

- `@Inject` o `@Resource`

Ciclo de vida de un Spring Bean

Aplicación:

- Dependencias de otros beans (pueden requerir que estén creados previamente)
- Tareas previas a la inicialización
- XML: `init-method` y `destroy-method`
- `@PostConstruct` y `@PreDestroy`
- `InitializingBean` y `DisposableBean`

init-method

```
<!-- DAO Beans -->
```

```
<bean id="bookDao" class="com.bookstore.data.BookDao"  
      p:hibernateTemplate-ref="hibernateTemplate" />
```

```
<bean id="accountDao" class="com.bookstore.data.AccountDao"  
      init-method="createTable"  
      p:jdbcTemplate-ref="jdbcTemplate" />
```

Nombre de método de la
clase **AccountDao**

destroy-method

```
<!-- DataSource Beans -->
```

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
```

```
  destroy-method="close"
```

```
    p:url="jdbc:hsqldb:file:database.dat;shutdown=true"
```

```
    p:driverClassName="org.hsqldb.jdbcDriver"
```

```
    p:username="sa"
```

```
    p:password="" />
```

Nombre de método de la clase
BasicDataSource

Fichero de propiedades (Configuración de properties)

- XML o annotation-based

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  
    <property name="locations" value="classpath:usuarios.properties" />  
</bean>
```

```
<context:property-placeholder location="classpath:pruebas.properties" />
```

Expression Language (SpEL)

- Usado en configuración **XML** o **annotation-based**
- En atributos miembro o parámetros de métodos
- **Syntax** para definir la expresión:

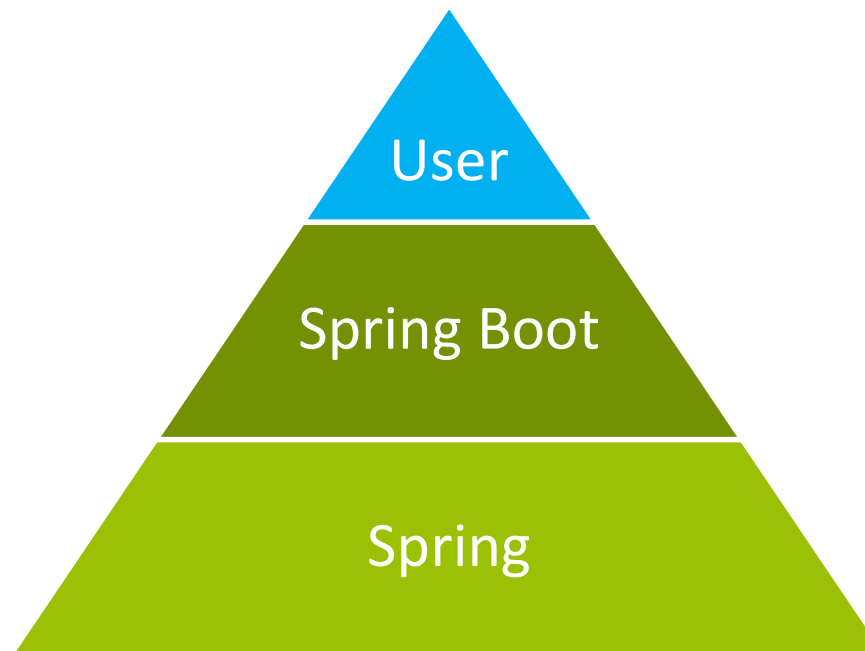
`#{ <expression string> }`

Spring

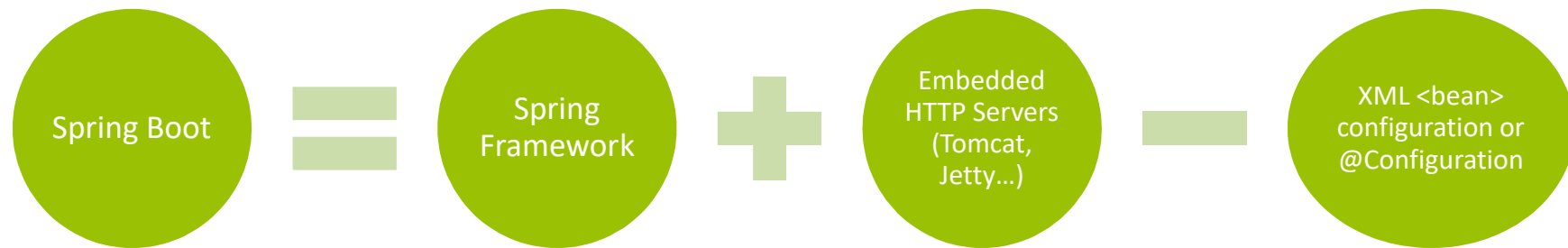
Spring Boot + Spring Initializr

Spring y Spring Boot

- Spring es un Framework de Java
- Spring Boot es un complemento de Spring que acelera la de creación de proyectos
- Con Spring Boot hacemos lo mismo que con Spring de una forma más ágil



Spring Boot

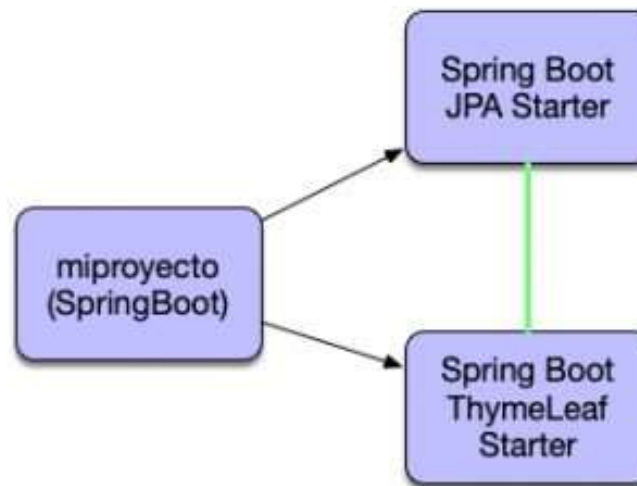


Spring Boot - Características

- Solución creada por Pivotal Software para desarrollar Microservicios en Java
- Basado en el principio de convención sobre configuración
- Reduce la complejidad del desarrollo de nuevos proyectos basado en Spring
- Incorporación directa de servidores web/contenedores como Apache Tomcat o Jetty, eliminando la necesidad de desplegar archivos WAR (Web Application Archive)
- Simplificación de la configuración de Maven gracias a los POM (Project Object Models) “starter”
- Configuración automática de Spring en la medida de lo posible
- Características no funcionales, como métricas o configuraciones externalizadas

Spring Boot Starter

- Simplificación y gestión correcta de dependencias
- Evita errores en la gestión de dependencias
- Definición de convenciones automáticas
- Solventar problemas en proyectos con otros frameworks integrados entre ellos
- Enfoque: generar dependencias ligadas con Spring de forma directa
- Mapea dependencias con sus versiones correspondientes



Spring Initializr - Características

- Servicio web para establecer la configuración de Spring Boot y después descargarla como plantilla de proyecto final
- Interfaz web fácil de usar que simplifica la creación de los JAR
- Utiliza Maven o Gradle para generar los archivos
- Debemos tener instalado Java + Maven o Gradle para trabajar en la aplicación Spring Boot

Project☒ Maven Project☐ Gradle Project**Language**☒ Java ☐ Kotlin☐ Groovy**Spring Boot**☐ 2.4 (SNAPSHOT) ☒ 2.3.1 ☐ 2.3.1 (SNAPSHOT)☐ 2.2.9 (SNAPSHOT) ☐ 2.2.8 ☐ 2.1.16 (SNAPSHOT)☐ 2.1.15**Project Metadata**

Group com.example

Artifact demo

Name demo

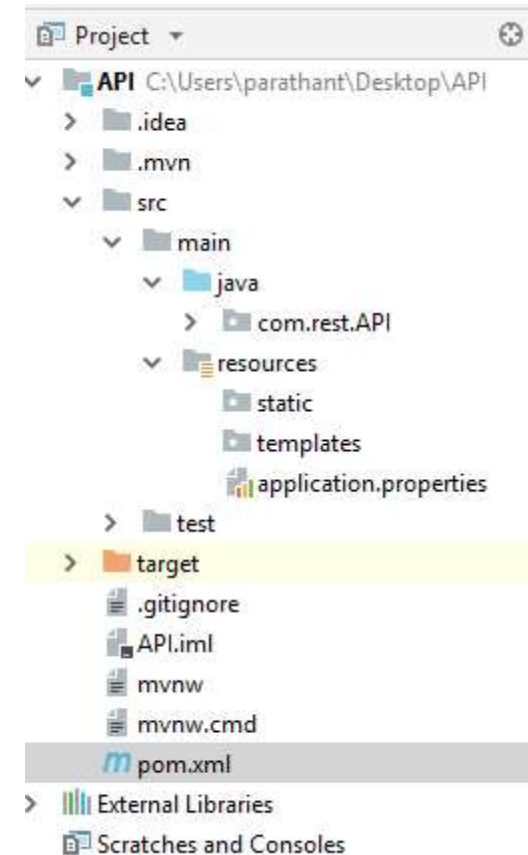
Description Demo project for Spring Boot

Package name com.example.demo

Packaging ☒ Jar ☐ WarJava ☐ 14 ☐ 11 ☒ 8**Dependencies****ADD DEPENDENCIES...** CTRL + B*No dependency selected*

Spring Initializr – Inicialización de un proyecto

- Ir a <https://start.spring.io/>
- Elegir opciones de configuración del proyecto
- Generar proyecto con alt + enter
- Descargamos proyecto generado en archivo .zip
- Descomprimir en directorio del proyecto



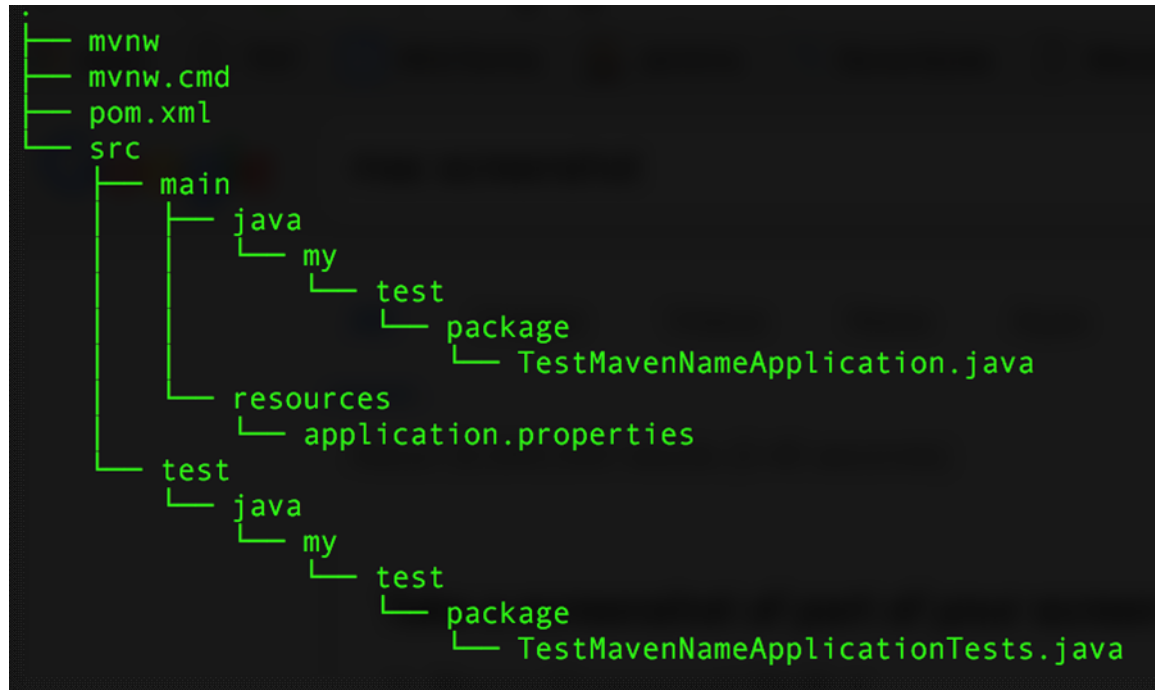
Spring Initializr – Opciones de configuración

- **Tipo de proyecto:** Maven o Gradle (artefacto pom.xml o build.gradle)
- **Versión de Spring Boot:** La versión del starter parent de Spring Boot de la que vamos a depender.
- **Group:** campo groupId en el descriptor de maven y nombre del paquete base de las clases de la aplicación
- **Artifact:** Nombre del artefacto. Maven: campos artifactId y name. Gradle: campo jar.baseName. Este será además el nombre del archivo zip que se va a generar
- **Dependencies:** buscador de dependencias que se corresponden con los starters de spring boot disponibles

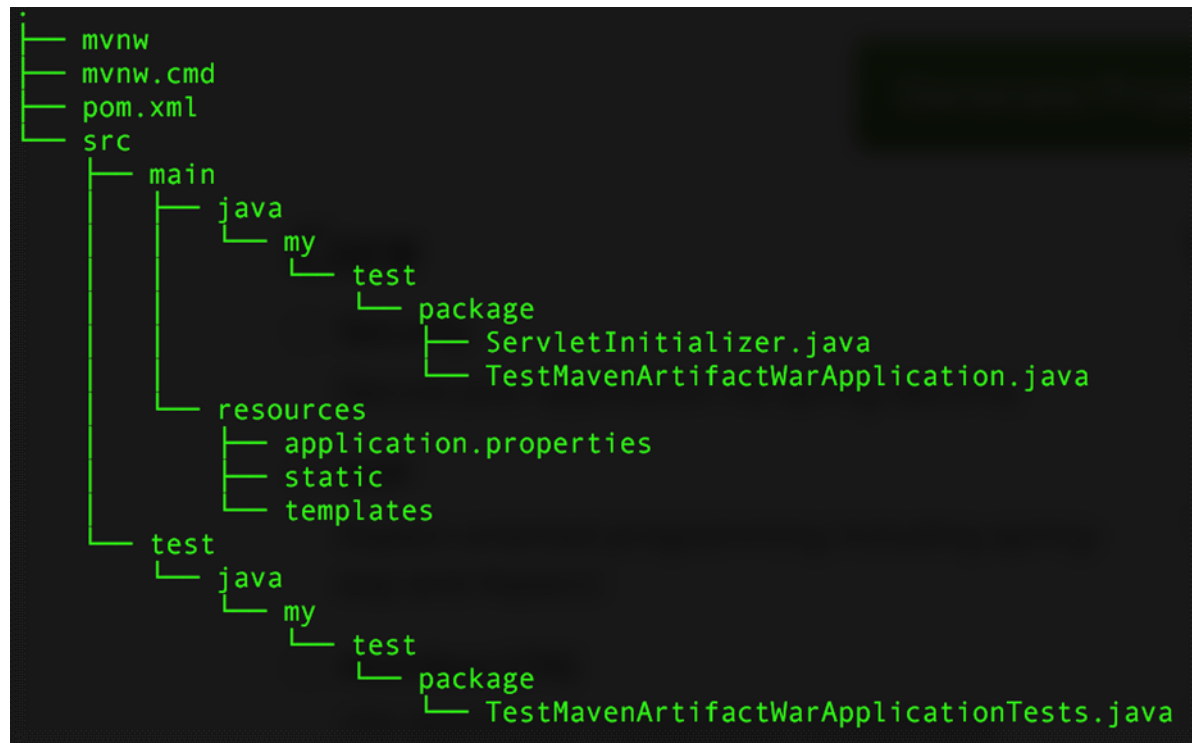
Spring Initializr – Opciones de configuración

- **Name:** campo name del archivo pom.xml. En gradle no tiene efecto.
- **Description:** campo description del archivo pom.xml. En gradle no tiene efecto.
- **Package Name:** Nombre del paquete base de las clases de la aplicación en caso de que sea diferente al campo Group.
- **Packaging:** Empaquetado del artefacto: jar o war. Esto afecta a los plugins de construcción que se especifican en el descriptor de la aplicación (pom.xml o build.gradle) y a la estructura de archivos que se crea.
- **Java Version:** Versión de java que vamos a especificar para nuestro artefacto.
Language: Java, Groovy y Kotlin.

Spring Initializr – Estructura generada con empaquetado JAR



Spring Initializr – Estructura generada con empaquetado WAR



Spring Initializr – Ejemplo clase principal Java generada

```
package com.rest.API;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class ApiApplication {
    public static void main(String[] args) {
        SpringApplication.run(ApiApplication.class, args);
    }
}
```

Módulo **SpringBootApplication** (org.springframework.boot)

- **@Component** tells the compiler that the following class is a component which should be included when compiling the whole application.
- **@ComponentScan** does the Scan of which packages we are going to use in the following Java class.
- **@EnableAutoConfiguration** enables Spring Boot's autoconfiguration mechanism to import important modules for the Spring Boot to run.
- **@SpringBootApplication** sustituye a **@Configuration**, **@ComponentScan** y **@EnableAutoConfiguration**

Spring Initializr – pom.xml generado (Maven)

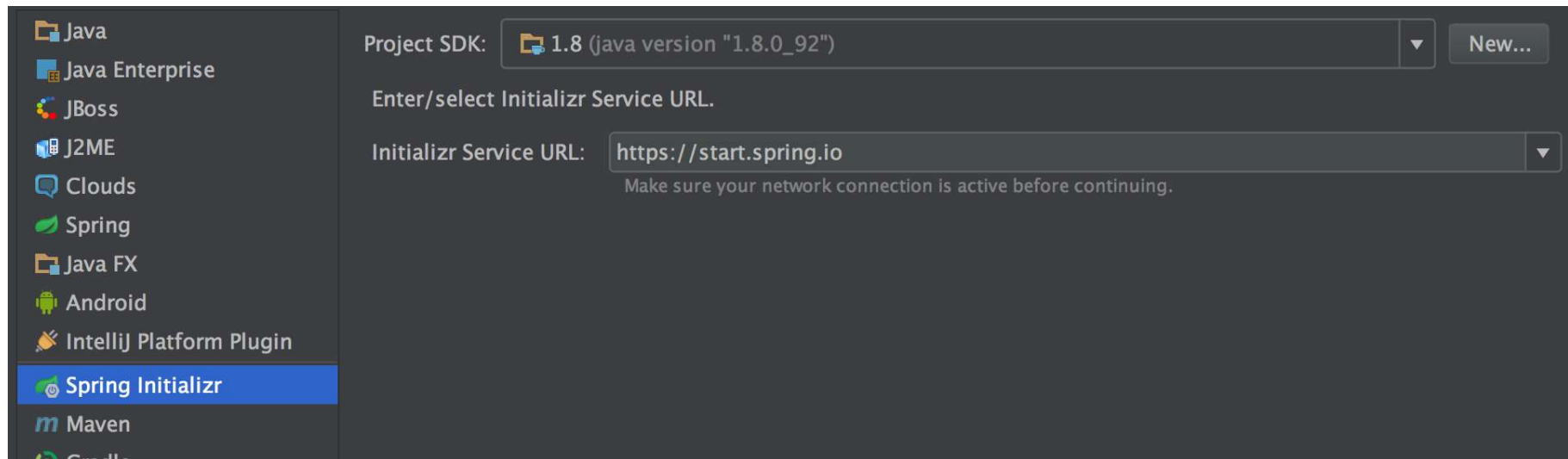
```
<groupId>com.example</groupId>
<artifactId>spring-boot</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>spring-boot</name>
<description>Demo project for Spring Boot</description>

<properties>
    <java.version>1.8</java.version>
</properties>

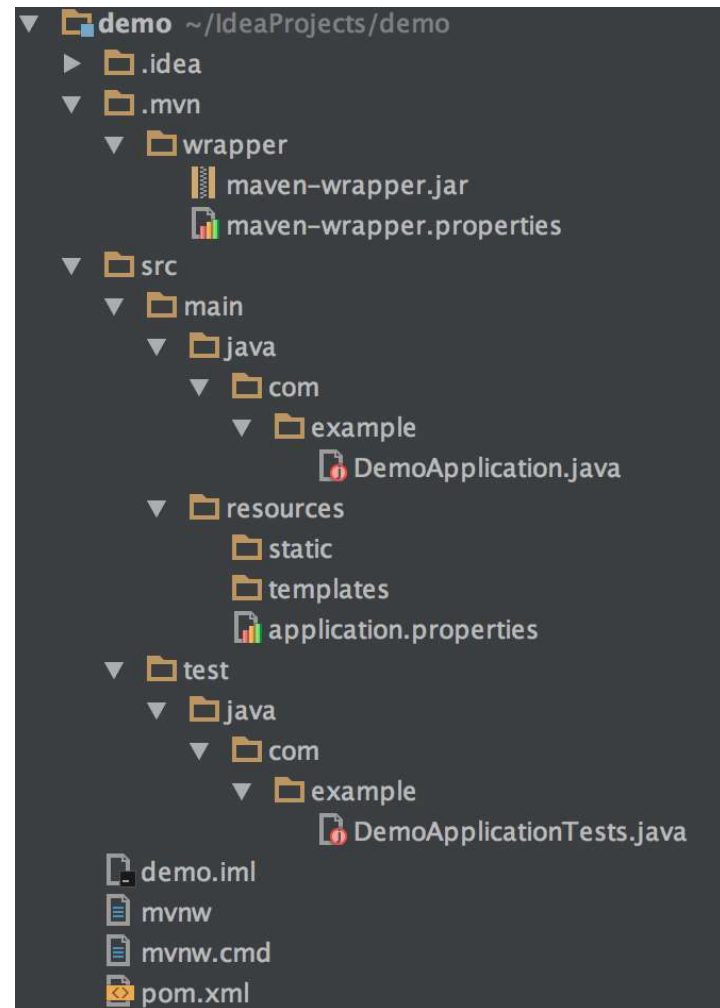
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
        <exclusions>
            <exclusion>
                <groupId>org.junit.vintage</groupId>
                <artifactId>junit-vintage-engine</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

Spring Initializr desde IntelliJ IDEA



Spring Initializr desde IntelliJ IDEA



Spring

Configuración de Spring Boot

Requisitos para trabajar con Spring Boot

- Java (JDK)
- Apache Maven
- IDE compatible (Spring Tool Suite, IntelliJ IDEA)

Spring Boot = Microservice

- Standalone Spring Apps
- Auto Configuration
- Embedded Servlet Container
- Make JAR Not WAR
- Production-Ready Features

Application.properties

- Archivo de configuración de los detalles de una aplicación Spring Boot
- Permite guardar y configurar principales propiedades* de entorno de la aplicación
- Ubicación: `src/main/resources`

Configuración de una Spring Boot Web Application | **Port Number**

- Cambiar puerto HTTP 80 por defecto

```
server.port=8083
```

- Basado en configuración YAML

```
server:  
  port: 8083
```

- Programáticamente

```
@Component  
public class CustomizationBean implements  
    WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {  
  
    @Override  
    public void customize(ConfigurableServletWebServerFactory container) {  
        container.setPort(8083);  
    }  
}
```

Configuración de una Spring Boot Web Application | **Context Path**

- Cambiar ruta “/” por defecto

```
server.servlet.contextPath=/springbootapp
```

- Basado en configuración YAML

```
server:
  servlet:
    contextPath: /springbootapp
```

- Programáticamente

```
@Component
public class CustomizationBean
    implements WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

    @Override
    public void customize(ConfigurableServletWebServerFactory container) {
        container.setContextPath("/springbootapp");
    }
}
```

Configuración de una Spring Boot Web Application

Configure the Logging Levels

- Tune the logging levels in a Boot application; Starting with version 1.2.0 onwards, you can configure the log level in the main properties file:

```
logging.level.org.springframework.web: DEBUG
```

```
logging.level.org.hibernate: ERROR
```

Configuración de una Spring Boot Web Application

Shut Down a Boot Application Programmatically

@Autowired

```
public void shutDown(ExecutorServiceExitCodeGenerator exitCodeGenerator) {  
    SpringApplication.exit(applicationContext, exitCodeGenerator);  
}
```

Configuración de una Spring Boot Web Application

Registrar un Servlet nuevo

```
@Bean
public SpringHelloServletRegistrationBean servletRegistrationBean() {


    SpringHelloServletRegistrationBean bean = new SpringHelloServletRegistrationBean(
        new SpringHelloWorldServlet(), "/springHelloWorld/*");
    bean.setLoadOnStartup(1);
    bean.addInitParameter("message", "SpringHelloWorldServlet special message");
    return bean;
}
```

Configuración de una Spring Boot Web Application

Configure Jetty in Boot Application

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

The Spring Boot starters generally use Tomcat as the default embedded server, you can exclude the Tomcat dependency and include Jetty instead (for example).

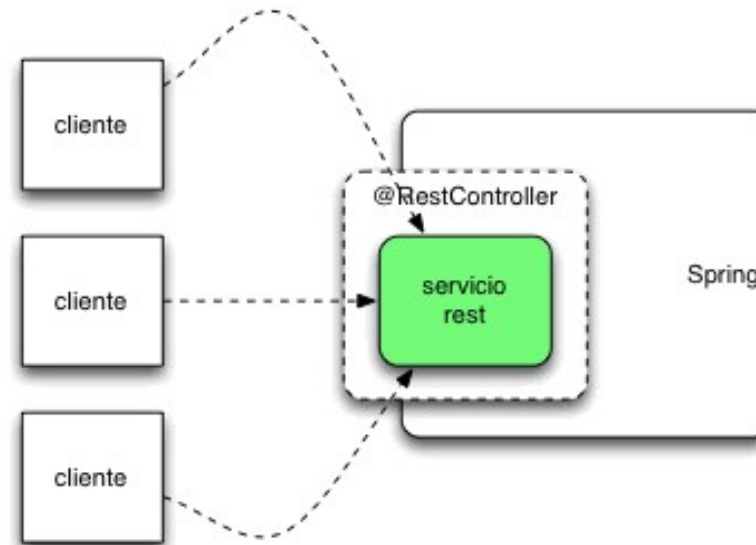


Spring

Configuración Entidad, Servicio, Controller

El Controlador - @RestController

- La clase anotada con `@RestController` será la encargada de gestionar las peticiones que se hagan a nuestra API.
- Indica que los datos devueltos por cada método se escribirán directamente en el cuerpo de la respuesta (response body).
- Sustituye al uso de `@Controller` + `@ResponseBody`.



El Servicio - **@Service**

- Funcionamiento parecido a **@Controller**
- Permite que Spring reconozca a la clase anotada como servicio al escanear los componentes de la aplicación

Implementación de la capa de Acceso a Datos

- Spring Boot + Spring data JPA

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <artifactId>spring-data-jpa</artifactId>
  <packaging>jar</packaging>
  <name>Spring Boot Spring Data JPA</name>
  <version>1.0</version>

  <parent>... </parent>

  <dependencies>
    <!-- jpa, crud repository -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
  </dependencies>
</project>
```

@Entity

- Anotación que define objeto para persistencia en bases de datos basadas en JPA
- Permite asociar una clase a una tabla o colección
- Otras implementaciones: Spring Data, MongoDB, Spring Data Cassandra, etc...
- Anotar clases del modelo de persistencia

La Entidad - @Entity

- Model and JPA annotations

@Entity

```
public class Customer {  
  
    // "customer_seq" is Oracle sequence name.  
    @Id  
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "CUST_SEQ")  
    @SequenceGenerator(sequenceName = "customer_seq", allocationSize = 1, name = "CUST_SEQ")  
    Long id;  
  
    String name;  
  
        String email;  
  
    @Column(name = "CREATED_DATE")  
    Date date;  
  
    //getters and setters, constructors  
}
```

Configuration + Database Initialization

- Configure **Oracle** data source

```
# Oracle settings
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=system
spring.datasource.password=password
spring.datasource.driver-class-oracle.jdbc.driver.OracleDriver
```



application.properties*

Configuration + Database Initialization

- Configure **MongoDB** data source

```
# Spring properties
spring:
  data:
    mongodb:
      host: localhost
      port: 27017
      uri: mongodb://localhost/test

# HTTP Server
server:
  port: 4444    # HTTP (Tomcat) port
```

application.yml*



Implementación de la capa de acceso a datos

- Spring Boot + Spring data JPA

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongo-java-driver</artifactId>
</dependency>
```


Implementación de la capa de acceso a datos

- Spring Data CrudRepository

```
import com.apirest.model.Customer;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;

import java.util.Date;
import java.util.List;
import java.util.stream.Stream;

public interface CustomerRepository extends CrudRepository<Customer, Long> {

    List<Customer> findByEmail(String email);

    List<Customer> findByDate(Date date);

    // custom query example and return a stream
    @Query("select c from Customer c where c.email = :email")
    Stream<Customer> findByEmailReturnStream(@Param("email") String email);

}
```

Recursos

<https://spring.io/guides/gs/spring-boot/>

<https://spring.io/docs>